The Improved GP 2 Compiler

Graham Campbell[1]*, Jack Romö[2], and Detlef Plump[3]

[1] School of Mathematics, Statistics and Physics, Newcastle University, Newcastle upon Tyne, UK
g.j.campbell2@newcastle.ac.uk
[2] Mathematical Institute, University of Oxford, Oxford, UK
jack.romo@lincoln.ox.ac.uk
[3] Department of Computer Science, University of York, York, UK
detlef.plump@york.ac.uk

Abstract. GP 2 is a rule-based programming language based on graph transformation rules which aims to facilitate program analysis and verification. Writing efficient programs in such a language is challenging because graph matching is expensive. GP 2 addresses this problem by providing rooted rules which, under mild conditions, can be matched in constant time. Recently, we implemented a number of changes to Bak's GP 2-to-C compiler in order to speed up graph programs. One key improvement is a new data structure for dynamic arrays called `BigArray`. This is an array of pointers to arrays of entries, successively doubling in size. To demonstrate the speed-up achievable with the new implementation, we present a reduction program for recognising binary DAGs which previously ran in quadratic time but now runs in linear time when compiled with the new compiler.

Keywords: Rule-Based Graph Programs · GP 2 · Time Complexity

1 Introduction

Rule-based graph transformation has been a topic of research since the early 1970s and has been the subject of numerous articles of both a theoretical and practical nature (see the monographs [11, 12]). In addition, various graph programming languages and model transformation tools have been developed, including AGG [22], GReAT [1], GROOVE [14], GrGen.Net [17], Henshin [2] and PORGY [13]. This paper focuses on the implementation of GP 2, an experimental programming language based on graph transformation rules [19]. GP 2 has been designed to support formal reasoning on programs [21, 20] and comes with a formal semantics in the style of structural operational semantics.

GP 2 programs manipulate directed graphs whose nodes and edges are labelled with heterogeneous lists of integers and character strings. The principal

* Supported by a Vacation Internship and a Doctoral Training Grant No. (2281162) from the Engineering and Physical Sciences Research Council (EPSRC) in the UK, while at University of York and Newcastle University, respectively.

programming construct in GP 2 are conditional graph transformation rules labelled with expressions. Rules operate on host graphs according to the attributed graph transformation framework of [16]: in a two-stage process, the rule is first instantiated by replacing all variables with values of the same type and evaluating all expressions, yielding a rule in the double-pushout approach with relabelling [15]. In the second stage, the instantiated rule is applied to the host graph by constructing two suitable pushouts.

The performance bottleneck for GP 2 (and graph transformation in general) is matching the left-hand graph L of a rule within a host graph G, requiring time polynomial in the size of L [4]. As a consequence, linear-time graph algorithms in imperative languages may be slowed down to polynomial time when they are recast as rule-based programs. To speed up matching, GP 2 supports *rooted* graph transformation where graphs in rules and host graphs are equipped with so-called root nodes, originally developed by Dörr [10]. Roots in rules must match roots in the host graph so that matches are restricted to the neighbourhood of the host graph's roots.

Using GP 2 programs with rooted rules, it has been possible to solve the 2-colouring problem for graphs of bounded degree in linear time [5]. Remarkably, the program generated by Bak's GP 2-to-C compiler matches the running time of Sedgewick's hand-crafted C implementation of the 2-colouring problem [23]. Meanwhile a few more conventional linear-time graph algorithms have been shown to have GP 2 implementations running in linear time on bounded-degree graphs [8]. To the best of our knowledge, no comparable performance results for low complexity graph algorithms have been reported for any other graph transformation language.

In this paper, we outline some performance issues of the existing GP 2-to-C compiler and the changes we made to address them, motivating each change, with examples where relevant. We also explain why we have not reached for off the shelf garbage collection algorithms. The new compiler produces programs with asymptotic runtime performance either the same as the original, or strictly faster. We give some timing results measuring the performance of the output programs of the original and improved compiler, providing empirical evidence both of our improvements and lack of performance degradation in cases beyond the scope of the improvements. We also implemented a root-reflecting mode to the compiler to allow a clean theoretical treatment of rooted rules.

2 Bak's GP 2 Compiler

Before we discuss our modifications to the GP 2 Compiler[4], we first outline its prior state. The compiler detailed in Bak's thesis [3] compiled GP 2 programs into C code, which was then compiled by the GCC compiler into an executable. The original compiler stored a graph as two dynamic arrays, one for nodes and one for edges. Internally, each of these contained two arrays, one of the actual elements

[4] The new compiler is available at https://github.com/UoYCS-plasma/GP2

and another of indices that were empty, or "holes". In the case of nodes, we dub these the node and hole arrays respectively. When iterating through nodes, each index would have to be checked to ensure it was not a hole. Deleting a node would require tracking the new hole and inserting nodes could be done by filling a hole should one exist. This raised performance issues if a program deleted a large number of nodes, for instance, in a graph reduction algorithm, as the enormous number of holes would make traversing the final smaller graph as slow as the original larger one.

We demonstrate this effect by the toy reduction program in Figure 1 which recognises edgeless graphs. (The loop `del!` applies the rule `del` as long as possible; the input graph is edgeless if and only if the program does not fail.) This program should run in linear time, as finding an arbitrary node should take constant time. Alas, each deleted node adds a new hole at the start of the node array, making the program take quadratic time due to traversing the holes at each rule match. The measured performance of both the original and improved implementation on discrete graphs is provided in Figure 2.

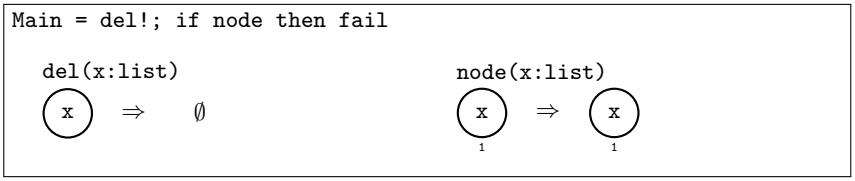

Fig. 1: The program `is-discrete.gp2`

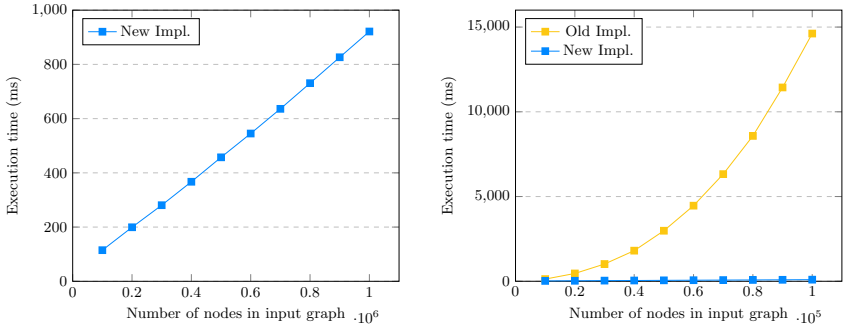

Fig. 2: Measured performance of `is-discrete.gp2`

Should the node array be too small for a new entry, it would be doubled with the `realloc()` C standard library function, the same being done for the hole array. However, this could change the array's position in memory, making any pointers to nodes invalid. This meant that Bak needed to store indices of nodes instead of pointers, adding extra memory operations to resolve indices every time a node was accessed.

For root nodes, Bak added a linked list to each graph, each entry holding a pointer to a root node in the graph's node array. Nodes would contain a

flag themselves detailing if they were a root node or not. Iterating through or deleting root nodes would now take constant time, should there be a constant upper bound on the number of root nodes.

A node itself contained its own index, the number of edges for which it is a source or target, termed its outdegree and indegree respectively, its incident edges, label, mark and several boolean flags. The incident edges to a node were stored in a unique manner, with indices of the first two edges being stored statically as part of the node type itself, and the rest in two dynamically allocated arrays of incoming and outgoing edges. This would avoid allocating arrays for a node should its degree be below three. Each edge would contain its own index, label, mark, and indices of its source and target nodes. It would, similarly to nodes, hold a boolean flag of whether it had been matched or not yet, to prevent overlapping matches.

Bak's compiler implementation included a parser for input graphs, necessary to allow programs to accept user provided input graphs. Unfortunately this implementation could not dynamically allocate memory to accommodate arbitrarily large input graphs. That is, only small inputs could be correctly processed, with memory overflows not gracefully handled or even necessarily detected.

Finally, to accommodate programs running within the condition of an `if` statement, for instance, a stack of states was needed. Bak implemented graph stacks in two varieties: copying the previous graph into a stack to reuse it when unwinding, and storing changes to the graph in the stack to undo them in reverse when unwinding. The former simply stored all the data that a node or edge contained, reconstructing the node or edge as needed as it unwound. Graph copying would simply perform a deep copy of the graph as expected.

3 The Improved Compiler

We have updated the compiler implementation to address the issues described in Section 2. We now present our improvements, which were first documented in the unpublished report [9]. We draw root nodes using double circles.

3.1 Graph Parsing

To resolve the issues with parsing, we decided to employ Judy arrays[5] [24], instead of a simple dynamic array. Invented by Doug Baskins, Judy arrays are a highly cache-optimised hash table implementation. The size of a Judy array is not statically pre-determined but is adjusted, at runtime, to accommodate the number of keys, which themselves can be integers or strings. Instead of storing nodes in the array directly, we now store pointers to nodes in the host graph as Judy arrays can only store references to a single word of data. Reallocating the array when doubling it could move the array around and invalidate previous pointers, an issue we resolve in the next subsection. This allowed an edge to

[5] http://judy.sourceforge.net

retrieve pointers to its source and target efficiently due to Judy arrays' fast runtime performance [18]. This also resolved problems with unnecessary node array size, allowing node IDs to be arbitrarily large without causing memory problems, as the array simply saw these IDs as meaningless keys. We consider these improvements to be bug fixes, rather than performance improvements, unlike our other internal data structure changes.

3.2 From Arrays to Linked Lists

To resolve the problems hole arrays pose, we switched to a linked list pointing to nodes. This change let us run the discrete graph deletion program in linear time, skipping holes we would otherwise traverse. Nodes, edges and now linked lists could then be stored in arrays doubling in size when needed as before. We choose this data structure above others that may have faster random access time, such as binary trees, because our only use case is iterating through the entire list to match subgraphs or adding and deleting nodes and edges. Accessing an arbitrary element of the data structure quickly is therefore irrelevant, meaning linked lists suit us perfectly.

It then became apparent that node indices were redundant, adding unneeded memory operations to resolve a node's address. Replacing indices with direct pointers would in turn add the problem of pointers being invalidated should the array of nodes or edges be moved when `realloc()` is called to enlarge them. To fix this, we replaced all internal arrays with a new type we dubbed `BigArray`. The `BigArray` type is an array of pointers to arrays of entries, successively doubling in size. Accessing a given index is constant time, using the position of the largest set bit in the index to identify which sub-array to access. A logarithmic number of memory allocations are performed overall when filling the array with entries, with the array of arrays being reallocated $O(\log(\log(n)))$ times. While such doubling arrays could lead to memory thrashing, in the case of GP 2 we are not interested in working with graphs of such a scale that this would typically happen.

Big arrays also manage holes like the prior implementation did. However, instead of using a second array of holes, big arrays store a linked list of holes within the hole entries in the array, keeping a pointer to the first hole. When a hole is created in the array, that position in the array is overwritten with the data of a new linked list entry, becoming the head of the list of holes. This avoids having to use extra memory for holes, making the `BigArray` type more memory efficient and making deletion of elements constant time.

Most importantly, big arrays allow one to allocate more memory to the array without having to possibly move previous entries in memory, simply creating a new array. This means the low number of memory allocations may be maintained without pointers to nodes and edges being invalidated. Thus, three big arrays are now stored within a graph, one for nodes, one for edges, and one for entries in the linked list of nodes, termed `NodeList`. A node list simply contains a pointer to the node it refers to and a pointer to the next entry in the linked list. The same is true of edge lists, albeit for edges.

Each node now contains a big array of linked list entries for edges and pointers to the linked list of outgoing edges and of incoming edges. No iteration through edges directly is ever needed beyond printing a graph, which can be done by iterating through the outgoing edges of every node, so no total list of edges is maintained. A list of edges is again acceptable: our only use case is iteration. Furthermore, as with the reasoning to add root nodes to GP 2, we are often interested in graphs of bounded degree, making fast random access of edges even less of a priority.

To avoid all pointers to a node or edge being garbage once it is deleted, nodes and edges were also made to track who references them, only being garbage collected when no such references exist. Nodes now hold flags representing if they are in a graph or referred to in the stack of graph changes, and edges remember if they are in a node's list of incoming/outgoing edges or in the stack also. Should a node or edge be deleted, the operation can be deferred should other references still exist. Now, stacks of graph changes can simply store pointers to nodes and edges rather than any data in them, as deletion would be automatic. We opted to use a more manual approach to garbage collection rather than existing implementations like the Boehm-Demers-Weiser garbage collector [6], as managing this ourselves massively reduces the amount of system calls needed and lets us even consider allocating memory in chunks doubling in size as we have done. Furthermore, garbage collection is not the core content of our improvements, as our focus remains on eliminating complexity issues in the aforementioned examples.

In the previous version of the compiler, there was only a single edge array associated with each node. Now each node has, in addition to an internal big array storing edges, two separate edge lists, for outgoing and incoming edges respectively. It is now possible to run programs that previously required bounded degree to obtain a certain worst case time complexity, now with only bounded incoming degree, or bounded outgoing degree, since search plans can now only consider edges of the correct orientation.

3.3 Root-Reflecting Mode

Finally, we have added a new "root-reflecting" mode to the compiler, allowing the programmer to decide if rule application should reflect and well as preserve root nodes. The motivation for this change is due to an issue in the theoretical foundation of rooted graph transformation with relabelling originally used by Bak and Plump which means that the right-hand square of a direct derivation need not be a natural pushout.

That is, in the usual definition of injective graph transformation, the application of a rule $L \hookleftarrow K \hookrightarrow R$ to a graph G, given a match $L \to G$, is the computation of the pushout complement $(D, K \to D, D \hookrightarrow G)$ of $L \to G$ and $K \hookrightarrow L$, if it exists, and then the pushout $(H, D \hookrightarrow H, R \to H)$ of $K \hookrightarrow R$ and $K \to D$. The pushout complement exists exactly when the dangling condition is satisfied, and D and H are unique up to unique isomorphism. When Plump and Habel introduced relabelling in 2002 [15], in order to guarantee uniqueness

they insisted on the pushout complement also being a pullback. This is no restriction, because both pushouts were already pullbacks in the original setting. By comparison, looking at Bak and Plump's explicit algorithm for computing derivations as the definition of rule application, we have two problems:

1. The right square need not be a pullback.
2. The left square need not be a pushout.

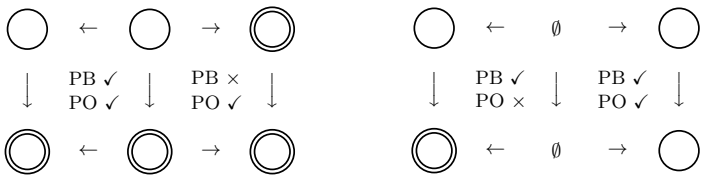

Fig. 3: Problematic Derivation Examples

The first problem could be argued to not be major, since we remain compatible with the original notion of injective DPO graph transformation, and we still have uniqueness. However, there are two unfortunate consequences. The first is that derivations are no longer invertible in the sense that if $G \Rightarrow_r H$, then $H \Rightarrow_{r^{-1}} G$. The second is that non-root-reflecting matches may not properly capure the intention of a programmer. Take for example, the left diagram of Figure 3. The intention of a programmer writing such a rule would have been to match a non-root and make it rooted, but if morphisms are not required to reflect rootedness of nodes, the effect of applying the rule could be to do nothing, if the non-root was matched against a root. Requiring root-reflecting morphisms excludes this case, and the rule would, instead, be non-applicable.

The second problem is more serious, as it represents an irrevocable incompatibility with the definition of DPO graph transformation. This problem was discovered by Plump and Wulandari in 2020, and their example is shown in the right diagram of Figure 3.

Fixing both these problems (at the time, only aware of the first problem), Campbell developed a new foundation for rooted GT systems with relabelling [7]. Instead of only insisting on matches preserving root nodes, one must additional insist on them reflecting them too. This was formalised by defining rootedness using a partial function into a two-point set rather than pointing graphs with root nodes, thus allowing both squares in a derivation to be natural pushouts, where formally, rules are allowed to have undefined rootedness in their interface graphs. Plump and Wulandari also adopt Campbell's new formalism in their recent work [25].

4 Timing Results

We ran various benchmarks, comparing the old with the new implementation, some examples of which are included here. Our full range of experiments can be found in [9], and the concrete syntax of the programs is also available[6].

[6] https://gist.github.com/GrahamCampbell/c8d84d42e3913065d1f9859fd8aeb8dd

In Section 2, we observed that even the program that simply deleted all isolated nodes could not be executed in linear time by the old implementation. A more subtle example is when the program splits up the input graph as it executes. In Figure 4, we give a rooted reduction program with this property that recognises binary directed acyclic graphs (DAGs).

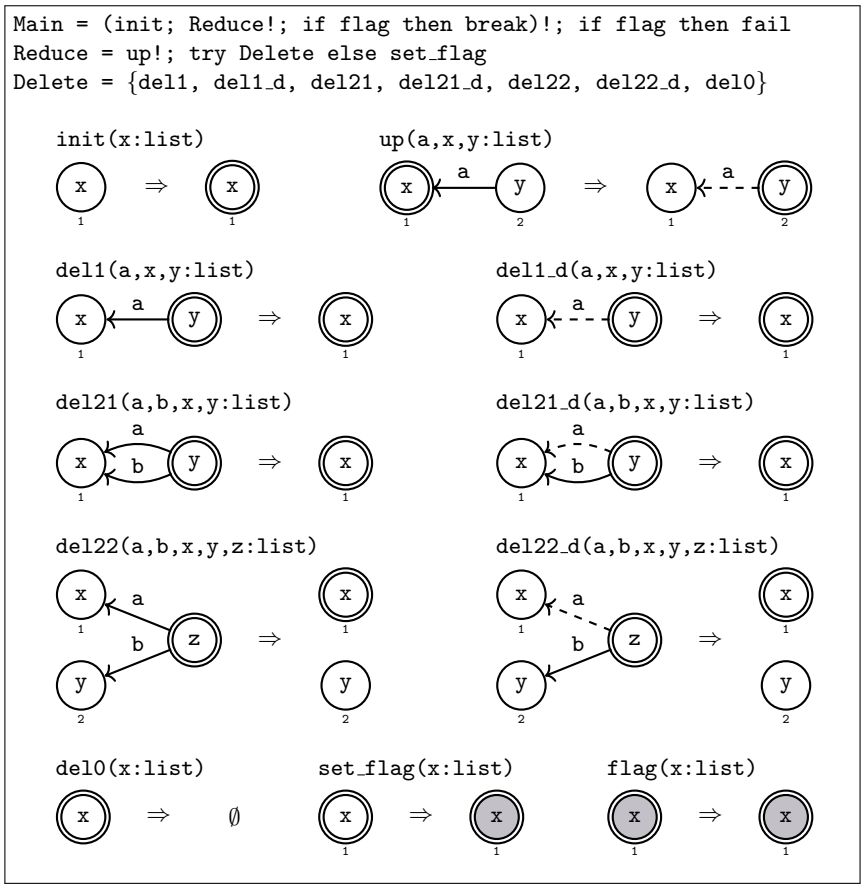

Fig. 4: The program `is-binary-dag.gp2`

The program implements a depth-first search for nodes without incoming edges, recording the "upward" path of the root node by dashed edges, combined with reduction steps that delete such nodes. The main loop either terminates with the empty graph, implying that the input graph was a binary DAG, or upon execution of the break statement when a node has been detected to which neither `up` nor one of the reduction rules is applicable. We observe that the original compiler produces a program that runs in quadratic time on many graph classes, including full binary trees and grid graphs. Our new compiler runs in linear time on such graphs (see Figure 5).

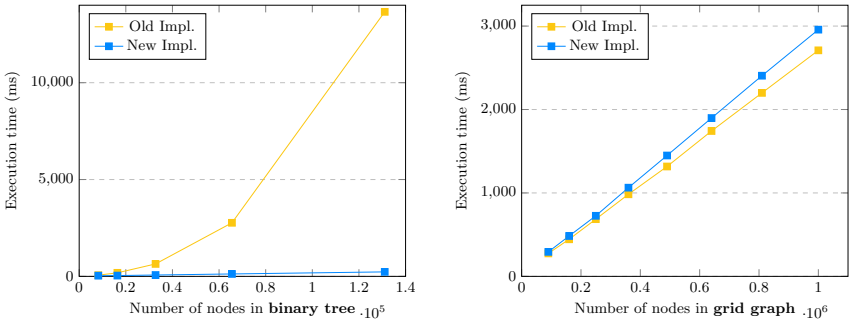

Fig. 5: Measured performance of `is-binary-dag.gp2`

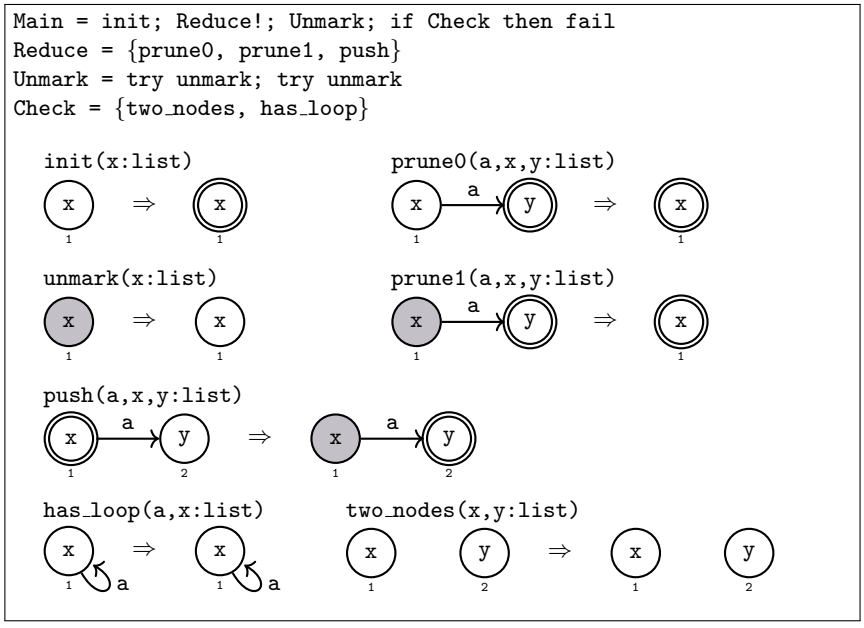

Fig. 6: GP 2 Program `is-tree.gp2`

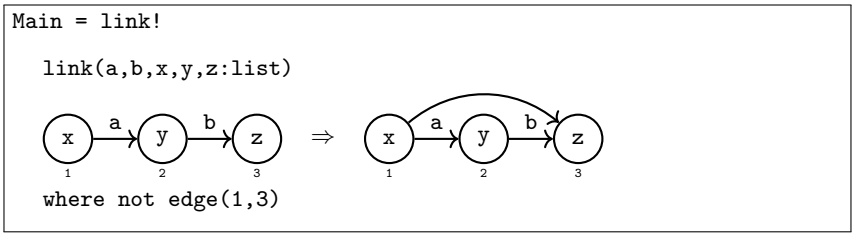

Fig. 7: The program `transitive-closure.gp2`

Next, we look at a rooted tree reduction program by Campbell [7] (Figure 6) that was linear time on graphs of bounded degree in the previous implementation of the compiler. We confirm that it remains linear time (Figure 8). Finally, we measure the performance of an unrooted program (Figure 7) which computes the transitive closure of a graph. The new compiler is superior on linked lists and grid graphs (Figure 9) due to the re-implementation of edge lists which speeds up the intense search for edges in the absence of root nodes.

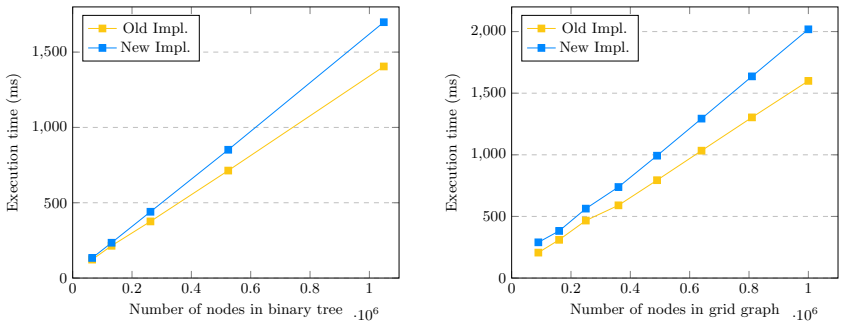

Fig. 8: Measured performance of `is-tree.gp2`

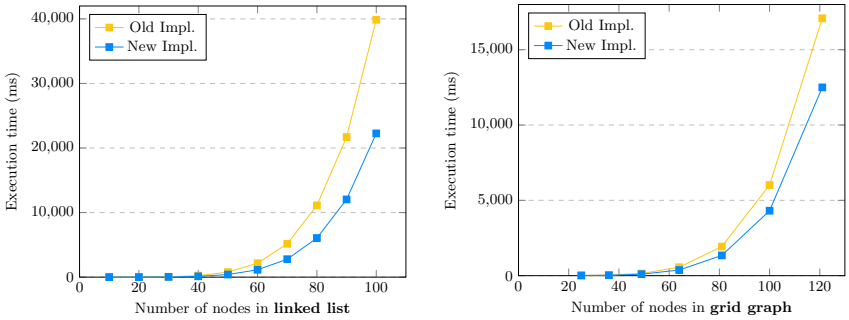

Fig. 9: Measured performance of `transitive-closure.gp2`

5 Conclusion and Future Work

The new compiler improves the performance of some programs significantly while retaining the complexity class for others. The program `is-discrete.gp2` has been brought down to linear complexity due to our node list's capacity to skip holes in the underlying node array. Moreover, the new compiler outperforms the old when running `transitive-closure.gp2` by a significant constant factor. We attribute this to better cache usage; in the old compiler, only the indices of the first two edges are statically stored in the `Node` type, whereas the new compiler stores several more direct pointers to nodes statically in a node's big array.

Some programs perform better or worse depending on the type of input graph. For example, `is-binary-dag.gp2` is accelerated from quadratic to linear time on binary trees and linked lists, but is slightly slower now on grid graphs. Such variance is to be expected with substantial implementation changes. Overall, the worst case drop in performance is by a constant factor, meaning no complexities were worsened in the observed test cases. Several programs perform similarly to before with a slightly enlarged constant, a byproduct of the memory operations that a linked-list data structure entails. Moreover, the new compiler resolves some fundamental bugs in the graph parser's implementation.

Future work should determine whether it makes sense to add lists of marked and unmarked nodes in order to find a marked or unmarked node in constant time. Ongoing research is also exploring what classes of graph algorithms can be implemented in linear time in GP 2 using the current compiler. Bak and Plump showed that a depth-first search of graphs of bounded degree and with a bounded number of connected components can be performed in linear time [4, 3], and also 2-colouring of such graphs. This paper shows that it is possible to execute a reduction algorithm for binary DAGs in linear time that doesn't limit the growth of the number of components.

References

1. Agrawal, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. Software & Systems Modeling **5**(3), 261–288 (2006). https://doi.org/10.1007/s10270-006-0027-7
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proc. 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010). Lecture Notes in Computer Science, vol. 6394, pp. 121–135 (2010). https://doi.org/10.1007/978-3-642-16145-2_9
3. Bak, C.: GP 2: Efficient Implementation of a Graph Programming Language. Ph.D. thesis, Department of Computer Science, University of York, UK (2015)
4. Bak, C., Plump, D.: Rooted graph programs. In: Proc. 7th International Workshop on Graph Based Tools (GraBaTs 2012) (2012). https://doi.org/10.14279/tuj.eceasst.54.780
5. Bak, C., Plump, D.: Compiling graph programs to C. In: Proc. 9th International Conference on Graph Transformation (ICGT 2016). Lecture Notes in Computer Science, vol. 9761, pp. 102–117 (2016). https://doi.org/10.1007/978-3-319-40530-8_7
6. Boehm, H.J., Demers, A., Weiser, M.: A garbage collector for C and C++. https://www.hboehm.info/gc/, accessed: 2020-05-01
7. Campbell, G.: Efficient Graph Rewriting. BSc thesis, Department of Computer Science, University of York, UK (2019), https://arxiv.org/abs/1906.05170
8. Campbell, G., Courtehoute, B., Plump, D.: Linear-time graph algorithms in GP 2. In: Proc. 8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 139, pp. 16:1–16:23 (2019). https://doi.org/10.4230/LIPIcs.CALCO.2019.16
9. Campbell, G., Romö, J., Plump, D.: Improving the GP 2 compiler. Tech. rep., Department of Computer Science, University of York (2019), https://arxiv.org/abs/2002.02914

10. Dörr, H.: Efficient Graph Rewriting and its Implementation, Lecture Notes in Computer Science, vol. 922. Springer (1995). https://doi.org/10.1007/BFb0031909
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2006). https://doi.org/10.1007/3-540-31188-2
12. Ehrig, H., Ermel, C., Golas, U., Hermann, F.: Graph and Model Transformation. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2015). https://doi.org/10.1007/978-3-662-47980-3
13. Fernández, M., Kirchner, H., Mackie, I., Pinaud, B.: Visual modelling of complex systems: Towards an abstract machine for PORGY. In: Proc. 10th Conference on Computability in Europe (CiE 2014). Lecture Notes in Computer Science, vol. 8493, pp. 183–193 (2014). https://doi.org/10.1007/978-3-319-08019-2_19
14. Ghamarian, A., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. International Journal on Software Tools for Technology Transfer **14**(1), 15–40 (2012). https://doi.org/10.1007/s10009-011-0186-x
15. Habel, A., Plump, D.: Relabelling in graph transformation. In: Proc. First International Conference on Graph Transformation (ICGT 2002). Lecture Notes in Computer Science, vol. 2505, pp. 135–147 (2002). https://doi.org/10.1007/3-540-45832-8_12
16. Hristakiev, I., Plump, D.: Attributed graph transformation via rule schemata: Church-Rosser theorem. In: Software Technologies: Applications and Foundations – STAF 2016 Collocated Workshops, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9946, pp. 145–160 (2016). https://doi.org/10.1007/978-3-319-50230-4_11
17. Jakumeit, E., Buchwald, S., Kroll, M.: GrGen.NET – the expressive, convenient and fast graph rewrite system. International Journal on Software Tools for Technology Transfer **12**(3–4), 263–271 (2010). https://doi.org/10.1007/s10009-010-0148-8
18. Luan, H., Du, X., Wang, S., Ni, Y., Chen, Q.: J^+-tree: A new index structure in main memory. In: Proc. 12th International Conference on Database Systems for Advanced Applications (DASFAA 2007). Lecture Notes in Computer Science, vol. 4443, pp. 386–397 (2007). https://doi.org/10.1007/978-3-540-71703-4_34
19. Plump, D.: The design of GP 2. In: Proc. 10th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011). pp. 1–16 (2012). https://doi.org/10.4204/EPTCS.82.1
20. Plump, D.: Reasoning about graph programs. In: Proc. 9th International Workshop on Computing with Terms and Graphs (TERMGRAPH 2016). pp. 35–44 (2016). https://doi.org/10.4204/EPTCS.225.6
21. Poskitt, C., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae **118**(1–2), 135–175 (2012). https://doi.org/10.3233/FI-2012-708
22. Runge, O., Ermel, C., Taentzer, G.: AGG 2.0 – new features for specifying and analyzing algebraic graph transformations. In: Proc. 4th International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011). Lecture Notes in Computer Science, vol. 7233, pp. 81–88 (2011). https://doi.org/10.1007/978-3-642-34176-2_8
23. Sedgewick, R.: Algorithms in C. Part 5: Graph Algorithms. Addison-Wesley, 3rd ed. edn. (2002)
24. Silverstein, A.: Judy iv shop manual. Tech. rep., Hewlett-Packard (2002), http://judy.sourceforge.net/application/shop_interm.pdf
25. Wulandari, G., Plump, D.: Verifying graph programs with first-order logic. In: Pre-Proc. 11th International Workshop on Graph Computation Models (GCM 2020) (2020), to appear